CS261: A Second Course in Algorithms
Lecture #3: The Push-Relabel Algorithm for Maximum Flow[*]

Tim Roughgarden[†]

January 12, 2016

1 Motivation

The maximum flow algorithms that we've studied so far are *augmenting path* algorithms, meaning that they maintain a flow and augment it each iteration to increase its value. In Lecture #1 we studied the Ford-Fulkerson algorithm, which augments along an arbitrary s-t path of the residual networks, and only runs in pseudopolynomial time. In Lecture #2 we studied the Edmonds-Karp specialization of the Ford-Fulkerson algorithm, where in each iteration a shortest s-t path in the residual network is chosen for augmentation. We proved a running time bound of $O(m^2 n)$ for this algorithm (as always, $m = |E|$ and $n = |V|$). Lecture #2 and Problem Set #1 discuss Dinic's algorithm, where each iteration augments the current flow by a blocking flow in a layered subgraph of the residual network. In Problem Set #1 you will prove a running time bound of $O(n^2 m)$ for this algorithm.

In the mid-1980s, a new approach to the maximum flow problem was developed. It is known as the "push-relabel" paradigm. To this day, push-relabel algorithms are often the method of choice in practice (even if they've never quite been the champion for the best worst-case asymptotic running time).

To motivate the push-relabel approach, consider the network in Figure 1, where k is a large number (like 100,000). Observe the maximum flow value is k. The Ford-Fulkerson and Edmonds-Karp algorithms run in $\Omega(k^2)$ time in this network. Moreover, much of the work feels wasted: each iteration, the long path of high-capacity edges has to be re-explored, even though it hasn't changed from the previous iteration. In this network, we'd rather route k units of flow from s to x (in $O(k)$ time), and then distribute this flow across the k paths from

[*]©2016, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

x to t (in $O(k)$ time, linear-time overall). This is the idea behind push-relabel algorithms.[1] Of course, if there were strictly less than k paths from x to t, then not all of the k units of flow can be routed from x to t, and the remainder must be resent to the source. What is a principled way to organize such a procedure in an arbitrary network?

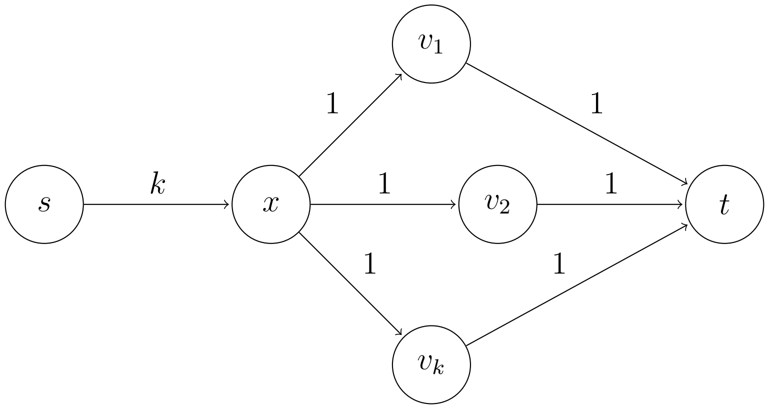


Figure 1: The edge $\{s, x\}$ has a large capacity k, and there are k paths from x to t via k different vertices v_i for $1 \leq i \leq k$ (3 are drawn for illustrative purposes). Both Ford-Fulkerson and Edmonds-Karp take $\Omega(k^2)$ time, but ideally we only need $O(k)$ time if we can somehow push k units of flow from s to x in one step.

2 Preliminaries

The first order of business is to relax the conservation constraints. For example, in Figure 1, if we've routed k units of flow to x but not yet distributed over the paths to t, then the vertex x has k units of flow incoming and zero units outgoing.

Definition 2.1 (Preflow) A *preflow* is a nonnegative vector $\{f_e\}_{e \in E}$ that satisfies two constraints:

Capacity constraints: $f_e \leq u_e$ for every edge $e \in E$;

Relaxed conservation constraints: for every vertex v other than s,

$$\text{amount of flow entering } v \geq \text{amount of flow exiting } v.$$

The left-hand side is the sum of the f_e's over the edge incoming to v; likewise with the outgoing edges for the right-hand side.

[1]The push-relabel framework is not the unique way to address this issue. For example, fancy data structures ("dynamic trees" and their ilk) can be used to remember the work performed by previous searches and obtain faster running times.

The definition of a preflow is exactly the same as a flow (Lecture #1), except that the conservation constraints have been relaxed so that the amount of flow into a vertex is allowed to exceed the amount of flow out of the vertex.

We define the residual graph G_f with respect to a preflow f exactly as we did for the case of a flow f. That is, for an edge e that carries flow f_e and capacity u_e, G_f includes a forward version of e with residual capacity $u_e - f_e$ and a reverse version of e with residual capacity f_e. Edges with zero residual capacity are omitted from G_f.

Push-relabel algorithms work with preflows throughout their execution, but at the end of the day they need to terminate with an actual flow. This motivates a measure of the "degree of violation" of the conservation constraints.

Definition 2.2 (Excess) For a flow f and a vertex $v \neq s, t$ of a network, the *excess* $\alpha_f(v)$ is

$$\text{amount of flow entering } v - \text{amount of flow exiting } v.$$

For a preflow flow f, all excesses are nonnegative. A preflow is a flow if and only if the excess of every vertex $v \neq s, t$ is zero. Thus transforming a preflow to recover feasibility involves reducing and eventually eliminating all excesses.

3 The Push Subroutine

How do we augment a preflow? When we were restricting attention to flows only, our hands were tied — to maintain the conservation constraints, we only augmented along an s-t (or, for a blocking flow, a collection of such paths). With the relaxed conservation constraints, we have much more flexibility. All we need to is to augment a flow along a single edge at a time, routing flow from one of its endpoints to the other.

```
Push($v$)

    choose an outgoing edge $(v, w)$ of $v$ in $G_f$ (if any)
    // push as much flow as possible
    let $\Delta = \min\{\alpha_f(v), \text{resid. cap. of } (v, w)\}$
    push $\Delta$ units of flow along $(v, w)$
```

The point of the second step is to send as much flow as possible from v to w using the edge (v, w) of G_f, subject to the two constraints that define a preflow. There are two possible bottlenecks. One is the residual capacity of the edge (v, w) (as dictated by nonnegativity/capacity constraints); if this binds, then the push is called *saturating*. The other is the amount of excess at the vertex v (as dictated by the relaxed conservation constraints); if this binds, the push is *non-saturating*. In the final step, the preflow is updated as in our augmenting path algorithms: if (v, w) the forward version of edge $e = (v, w)$ in G, then f_e is increased by Δ; if (v, w) the reverse version of edge $e = (w, v)$ in G, then f_e is decreased by Δ. As always, the residual network is then updated accordingly. Note that after pushing flow from v to w, w has positive excess (if it didn't already).

4 Heights and Invariants

Just pushing flow around the residual network is not enough to obtain a correct maximum flow algorithm. One worry is illustrated by the graph in Figure 2 — after initially pushing one unit flow from s to v, how do we avoid just pushing the excess around the cycle $v \to w \to x \to y \to v$ forevermore. Obviously we want to push the excess to t when it gets to x, but how can we be systematic about it?

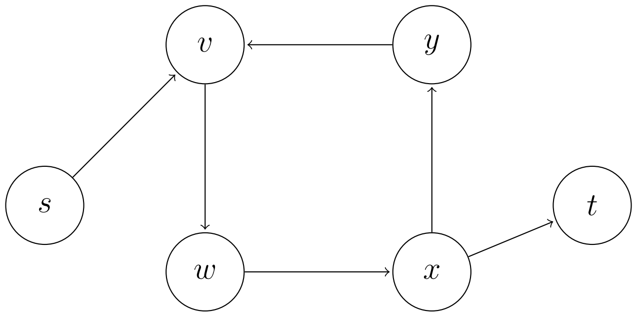


Figure 2: When we push flows in the above graph, how do we ensure that we do not push flows in the cycle $v \to w \to x \to y \to v$?

The next key idea will ensure termination of our algorithm, and will also implies correctness as termination. The idea is to maintain a *height* $h(v)$ for each vertex v of G. Heights will always be nonnegative integers. You are encouraged to visualize a network in 3D, with the height of a vertex giving it its z-coordinate, with edges going "uphill" and "downhill," or possibly staying flat. The plan for the algorithm is to always maintain three invariants (two trivial and one non-trivial):

Invariants

1. $h(s) = n$ at all times (where $n = |V|$);
2. $h(t) = 0$;
3. for every edge (v, w) of the current residual network (with positive residual capacity), $h(v) \leq h(w) + 1$.

Visually, the third invariant says that edges of the residual network are only to go downhill gradually (by one per hop). For example, if a vertex v has three outgoing edges (v, w_1), (v, w_2), and (v, w_3), with $h(w_1) = 3$, $h(w_2) = 4$, and $h(w_3) = 6$, then the third invariant requires that $h(v)$ be 4 or less (Figure 3). Note that edges are allowed to go uphill, stay flat, or go downhill (gradually).

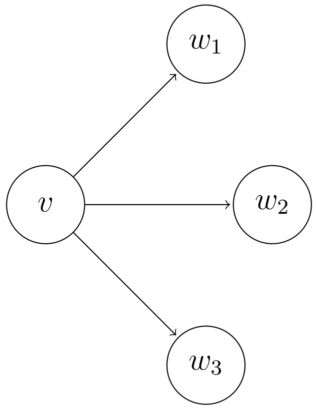


Figure 3: Given that $h(w_1) = 3, h(w_2) = 4, h(w_3) = 6$, it must be that $h(v) \leq 4$.

Where did these invariants come from? For one motivation, recall from Lecture #2 our optimality conditions for the maximum flow problem: a flow is maximum if and only if there is no s-t path (with positive residual capacity) in its residual graph. So clearly we want this property at termination. The new idea is to satisfy the optimality conditions *at all times*, and this is what the invariants guarantee. Indeed, since the invariants imply that s is at height n, t is at height 0, and each edge of the residual graph only goes downhill by at most 1, there can be no s-t path with at most $n - 1$ edges (and hence no s-t path at all). It follows that if we find a preflow that is feasible (i.e., is actually a flow, with no excesses) and the invariants hold (for suitable heights), then the flow must be a maximum flow.

It is illuminating to compare and contrast the high-level strategies of augmenting path algorithms and of push-relabel algorithms.

Augmenting Path Strategy

Invariant: maintain a feasible flow.

Work toward: disconnecting s and t in the current residual network.

Push-Relabel Strategy

Invariant: maintain that s, t disconnected in the current residual network.

Work toward: feasibility (i.e., conservation constraints).

While there is a clear symmetry between the two approaches, most people find it less intuitive to relax feasibility and only restore it at the end of the algorithm. This is probably why the push-relabel framework only came along in the 1980s, while the augmenting path algorithms we studied date from the 1950s-1970s. The idea of relaxing feasibility is useful for many different problems.

In both cases, algorithm design is guided by an explicitly articulated strategy for guaranteeing correctness. The maximum flow problem, while polynomial-time solvable (as we know), is complex enough that solutions require significant discipline. Contrast this with, for example, the minimum spanning tree algorithms, where it's easy to come up with correct algorithms (like Kruskal or Prim) without any advance understanding of *why* they are correct.

5 The Algorithm

The high-level strategy of the algorithm is to maintain the three invariants above while trying to zero out any remaining excesses. Let's begin with the initialization. Since the invariants reference both a correct preflow and current vertex heights, we need to initialize both. Let's start with the heights. Clearly we set $h(s) = n$ and $h(t) = 0$. The first non-trivial decision is to set $h(v) = 0$ also for all $v \neq s, t$. Moving onto the initial preflow, the obvious idea is to start with the zero flow. *But this violates the third invariant:* edges going out of s would travel from height n to 0, while edges of the residual graph are supposed to only go downhill by 1. With the current choice of height function, no edges out of s can appear (with non-zero capacity) in the residual network. So the obvious fix is to initially saturate all such edges.

Initialization

```
set $h(s) = n$
set $h(v) = 0$ for all $v \neq s$
set $f_e = u_e$ for all edges $e$ outgoing from $s$
set $f_e = 0$ for all other edges
```

All three invariants hold after the initialization (the only possible violation is the edges out of s, which don't appear in the initial residual network). Also, f is initialized to a preflow (with flow in $\geq$ flow out except at s).

Next, we restrict the Push operation from Section 3 so that it maintains the invariants. The restriction is that *flow is only allowed to be pushed downhill in the residual network.*

Push(v) [revised]

```
choose an outgoing edge $(v, w)$ of $v$ in $G_f$ with $h(v) = h(w) + 1$ (if any)
// push as much flow as possible
let $\Delta = \min\{\alpha_f(v), \text{resid. cap. of } (v, w)\}$
push $\Delta$ units of flow along $(v, w)$
```

Here's the main loop of the push-relabel algorithm:

Main Loop

```
while there is a vertex $v \neq s, t$ with $\alpha_f(v) > 0$ do
    choose such a vertex $v$ with the maximum height $h(v)$
    // break ties arbitrarily
    if there is an outgoing edge $(v, w)$ of $v$ in $G_f$ with $h(v) = h(w) + 1$
    then
        Push($v$)
    else
        increment $h(v)$                    // called a ``relabel''
```

Every iteration, among all vertices that have positive excess, the algorithm processes the highest one. When such a vertex v is chosen, there may or may not be a downhill edge emanating from v (see Figure 4(a) vs. Figure 4(b)). Push(v) is only invoked if there is such an edge (in which case Push will push flow on it), otherwise the vertex is "relabeled," meaning its height is increased by one.

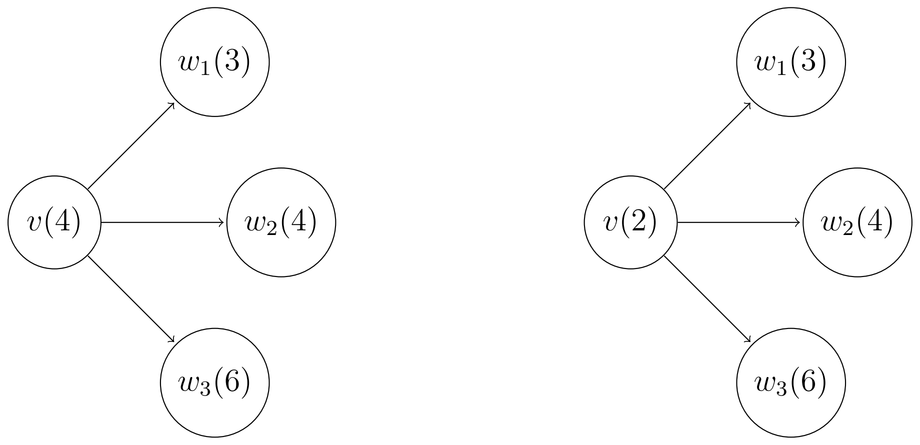


Figure 4: (a) $v \to w_1$ is downhill edge (4 to 3) (b) there are no downhill edges

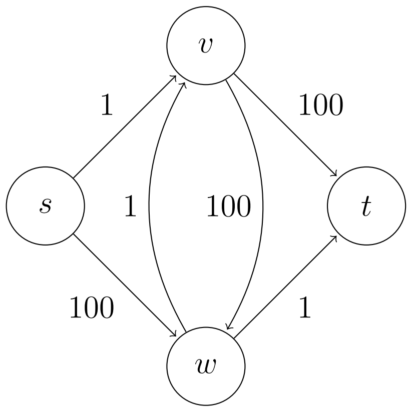
Lemma 5.1 (Invariants Are Maintained) *The three invariants are maintained throughout the execution of the algorithm.*

Neither s not t ever get relabeled, so the first two invariants are always satisfied. For the third invariant, we consider separately a relabel (which changes the height function but not the preflow) and a push (which changes the preflow but not the height function). The only worry with a relabel at v is that, afterwards, some outgoing edge of v on the residual network goes downhill by more than one step. But the precondition for relabeling is that all outgoing edges are either flat or uphill, so this never happens. The only worry with a push from v to w is that it could introduce a new edge (w, v) to the residual network that might go downhill by more than one step. But we only push flow downward, so a newly created reverse edge can only go upward.

The claim implies that if the push-relabel algorithm ever terminates, then it does so with a maximum flow. The invariants imply the maximum flow optimality conditions (no s-t path in the residual network), while the termination condition implies that the final preflow f is in fact a feasible flow.

6 Example

Before proceeding to the running time analysis, let's go through an example in detail to make sure that the algorithm makes sense. The initial network is shown in Figure 5(a). After the initialization (of both the height function and the preflow) we obtain the residual network in Figure 5(b). (Edges are labeled with their residual capacities, vertices with both their heights and their excesses.) [2]



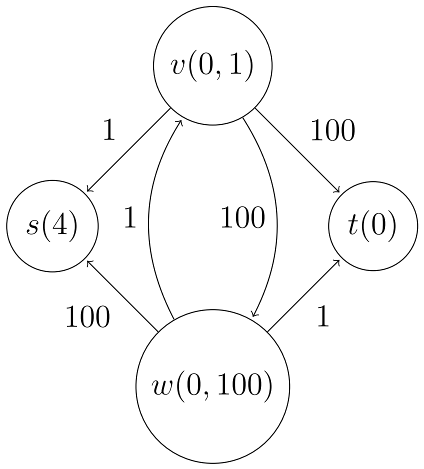


Figure 5: (a) Example network (b) Network after initialization. For v and w, the pair (a, b) denotes that the vertex has height a and excess b. Note that we ignore excess of s and t, so s and t both only have a single number denoting height.

In the first iteration of the main loop, there are two vertices with positive excess (v and w), both with height 0, and the algorithm can choose arbitrarily which one to process. Let's process v. Since v currently has height 0, it certainly doesn't have any outgoing edges in the residual network that go down. So, we relabel v, and its height increases to 1. In the second iteration of the algorithm, there is no choice about which vertex to process: v is now the unique highest label with excess, so it is chosen again. Now v *does* have downhill outgoing

[2]We looked at this network last lecture and determined that the maximum flow value is 3. So we should be skeptical of the 100 units of flow currently on edge (s, w); it will have to return home to roost at some point.

edges, namely (v,w) and (v,t). The algorithm is allowed to choose arbitrarily between such edges. You're probably rooting for the algorithm to push v's excess straight to t, but to keep things interesting let's assume that that the algorithm pushes it to w instead. This is a non-saturating push, and the excess at v drops to zero. The excess at w increases from 100 to 101. The new residual network is shown in Figure 6.

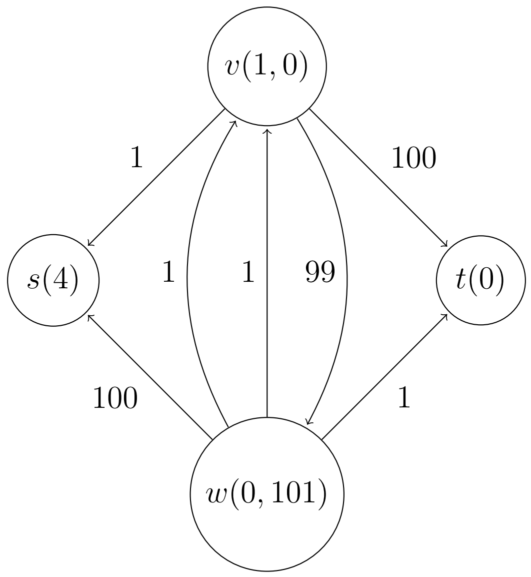


Figure 6: Residual network after non-saturating push from v to w.

In the next iteration, w is the only vertex with positive excess so it is chosen for processing. It has no outgoing downhill edges, so it get relabeled (so now $h(w) = 1$). Now w does have a downhill outgoing edge (w,t). The algorithm pushes one unit of flow on (w,t) — a saturating push —- the excess at w goes back down to 100. Next iteration, w still has excess but has no downhill edges in the new residual network, so it gets relabeled. With its new height of 2, in the next iteration the edges from w to v go downhill. After pushing two units of flow from w to v — one on the original (w,v) edge and one on the reverse edge corresponding to (v,w) — the excess at w drops to 98, and v now again has an excess (of 2). The new residual network is shown in Figure 7.

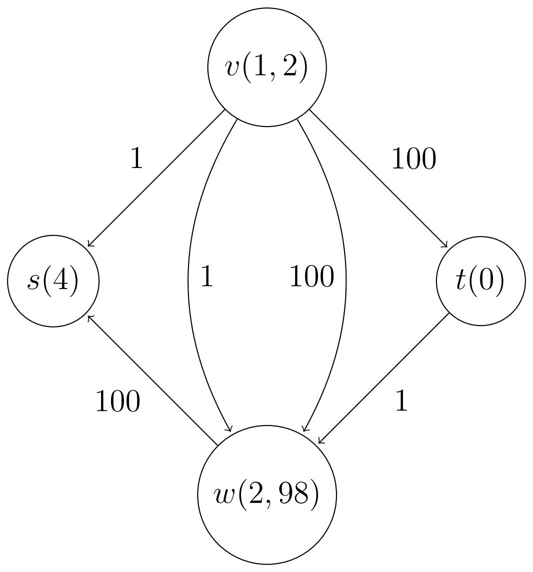


Figure 7: Residual network after non-saturating push from v to w.

Of the two vertices with excess, w is higher. It again has no downhill edges, however, so the algorithm relabels it three times in a row until it does. When its height reaches 5, the reverse edge (v, s) goes downhill, the algorithm pushes w's entire excess to s. Now v is the only vertex remaining with excess. Its edge (v, t) goes down hill, and after pushing two units of flow on it the algorithm halts with a maximum flow (with value 3).

7 The Analysis

7.1 Formal Statement and Discussion

Verifying that the push-relabel algorithm computes a maximum flow in one particular network is all fine and good, but it's not at all clear that it is correct (or even terminates) in general. Happily, the following theorem holds.[3]

Theorem 7.1 *The push-relabel algorithm terminates after $O(n^2)$ relabel operations and $O(n^3)$ push operations.*

The hidden constants in Theorem 7.1 are at most 2. Properly implemented, the push-relabel algorithm has running time $O(n^3)$; we leave the details to Exercise Set #2. The one point that requires some thought is to maintain suitable data structures so that a highest vertex with excess can be identified in $O(1)$ time.[4] In practice, the algorithm tends to run in sub-quadratic time.

[3]A sharper analysis yields the better bound of $O(n^2\sqrt{m})$; see Problem Set #1. Believe it or now, the worst-case running time of the algorithm is in fact $\Omega(n^2\sqrt{m})$.

[4]Or rather, $O(1)$ "amortized" time, meaning in total time $O(n^3)$ over all of the $O(n^3)$ iterations.

The proof of Theorem 7.1 is more indirect then our running time analyses of augmenting path algorithms. In the latter algorithms, there are clear progress measures that we can use (like the difference between the current and maximum flow values, or the distance between s and t in the current residual network). For push-relabel, we require less intuitive progress measures.

7.2 Bounding the Relabels

The analysis begins with the following key lemma, proved at the end of the lecture.

Lemma 7.2 (Key Lemma) *If the vertex v has positive excess in the preflow f, then there is a path $v \rightsquigarrow s$ in the residual network G_f.*

The intuition behind the lemma is that, since the excess for to v somehow from v, it should be possible to "undo" this flow in the residual network.

For the rest of this section, we assume that Lemma 7.2 is true and use it to prove Theorem 7.1. The lemma has some immediate corollaries.

Corollary 7.3 (Height Bound) *In the push-relabel algorithm, every vertex always has height at most $2n$.*

Proof: A vertex v is only relabeled when it has excess. Lemma 7.2 implies that, at this point, there is a path from v to s in the current residual network G_f. There is therefore such a path with at most $n-1$ edges (more edges would create a cycle, which can be removed to obtain a shorter path). By the first invariant (Section 4), the height of s is always n. By the third invariant, edges of G_f can only go downhill by one step. So traversing the path from v to s decreases the height by at most $n-1$, and winds up at height n. Thus v has height $2n-1$ or less, and at most one more than this after it is relabeled for the final time. ■

The bound in Theorem 7.1 on the number of relabels follows immediately.

Corollary 7.4 (Relabel Bound) *The push-relabel algorithm performs $O(n^2)$ relabels.*

7.3 Bounding the Saturating Pushes

We now bound the number of pushes. We piggyback on Corollary 7.4 by using the number of relabels as a progress measure. We'll show that lots of pushes happen only when there are already lots of relabels, and then apply our upper bound on the number of relabels.

We handle the cases of saturating pushes (which saturate the edge) and non-saturating pushes (which exhaust a vertex's excess) separately.[5] For saturating pushes, think about a particular edge (v, w). What has to happen for this edge to suffer two saturating pushes in the same direction?

[5] To be concrete, in case of a tie let's call it a non-saturating push.

Lemma 7.5 (Saturating Pushes) *Between two saturating pushes on the same edge (v, w) in the same direction, each of v, w is relabeled at least twice.*

Since each vertex is relabeled $O(n)$ times (Corollary 7.3), each edge (v, w) can only suffer $O(n)$ saturating pushes. This yields a bound of $O(mn)$ on the number of saturating pushes. Since $m = O(n^2)$, this is even better than the bound of $O(n^3)$ that we're shooting for.[6]

Proof of Lemma 7.5: Suppose there is a saturating push on the edge (v, w). Since the push-relabel algorithm only pushes downhill, v is higher than w ($h(v) = h(w) + 1$). Because the push saturates (v, w), the edge drops out of the residual network. Clearly, a prerequisite for another saturating push on (v, w) is for (v, w) to reappear in the residual network. The only way this can happen is via a push in the opposite direction (on (w, v)). For this to occur, w must first reach a height larger than that of v (i.e., $h(w) > h(v)$), which requires w to be relabeled at least twice. After (v, w) has reappeared in the residual network (with $h(v) < h(w)$), no flow will be pushed on it until v is again higher than w. This requires at least two relabels to v. ■

7.4 Bounding the Non-Saturating Pushes

We now proceed to the non-saturating pushes. Note that nothing we've said so far relies on our greedy criterion for the vertex to process in each iteration (the highest vertex with excess). This feature of the algorithm plays an important role in this final step.

Lemma 7.6 (Non-Saturating Pushes) *Between any two relabel operations, there are at most n non-saturating pushes.*

Corollary 7.4 and Lemma 7.6 immediately imply a bound of $O(n^3)$ on the number of non-saturating pushes, which completes the proof of Theorem 7.1 (modulo the key lemma).

Proof of lemma 7.6: Think about the entire sequence of operations performed by the algorithm. "Zoom in" to an interval bracketed by two relabel operations (possibly of different vertices), with no relabels in between. Call such an interval a *phase* of the algorithm. See Figure 8.

[6] We're assuming that the input network has no parallel edges, between the same pair of vertices and in the same direction. This is effectively without loss of generality — multiple edges in the same direction can be replaced by a single one with capacity equal to the sum of the capacities of the parallel edges.

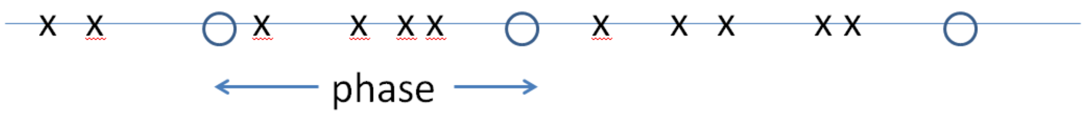


Figure 8: A timeline showing all operations ('O' represents relabels, 'X' represents non-saturating pushes). An interval between two relabels ('O's) is called a phase. There are $O(n^2)$ phases, and each phase contains at most n non-saturating pushes.

How does a non-saturating push at a vertex v make progress? By zeroing out the excess at v. Intuitively, we'd like to use the number of zero-excess vertices as a progress measure within a phase. But a non-saturating push can create a new excess elsewhere. To argue that this can't go on for ever, we use that *excess is only transferred from higher vertices to lower vertices.*

Formally, by the choice of v, as the highest vertex with excess, we have

$$h(v) \geq h(w) \qquad \text{for all vertices } w \text{ with excess} \tag{1}$$

at the time of a non-saturating push at v. Inequality (1) continues to hold as long as there is no relabel: pushes only send flow downhill, so can only transfer excess from higher vertices to lower vertices.

After the non-saturating push at v, its excess is zero. How can it become positive again in the future?[7] It would have to receive flow from a higher vertex (with excess). This cannot happen as long as (1) holds, and so can't happen until there's a relabel. We conclude that, within a phase, there cannot be two non-saturating pushes at the same vertex v. The lemma follows. ■

7.5 Analysis Recap

The proof of Theorem 7.1 has several cleverly arranged steps.

1. Each vertex can only be relabeled $O(n)$ times (Corollary 7.3 via Lemma 7.2), for a total of $O(n^2)$ relabels.
2. Each edge can only suffer $O(n)$ saturating pushes (only 1 between each time both endpoints are relabeled twice, by Lemma 7.5)), for a total of $O(mn)$ saturating pushes.
3. Each vertex can only suffer $O(n^2)$ non-saturating pushes (only 1 per phase, by Lemma 7.6), for a total of $O(n^3)$ such pushes.

[7]For example, recall what happened to the vertex v in the example in Section 6.

8 Proof of Key Lemma

We now prove Lemma 7.2, that there is a path from every vertex with excess back to the source s in the residual network. Recall the intuition: excess got to v from s somehow, and the reverse edges should form a breadcrumb trail back to s.

Proof of Lemma 7.2: Fix a preflow f.[8] Define

$$A = \{v \in V \ : \text{ there is an } s \rightsquigarrow v \text{ path } P \text{ in } G \text{ with } f_e > 0 \text{ for all } e \in P\}.$$

Conceptually, run your favorite graph search algorithm, starting from s, in the subgraph of G consisting of the edges that carry positive flow. A is where you get stuck. (This is the second example we've seen of the "reachable vertices" proof trick; there are many more.)

Why define A? Note that for a vertex $v \in A$, there is a path of reverse edges (with positive residual capacity) from v to s in the residual network G_f. So we just have to prove that all vertices with excess have to be in A.

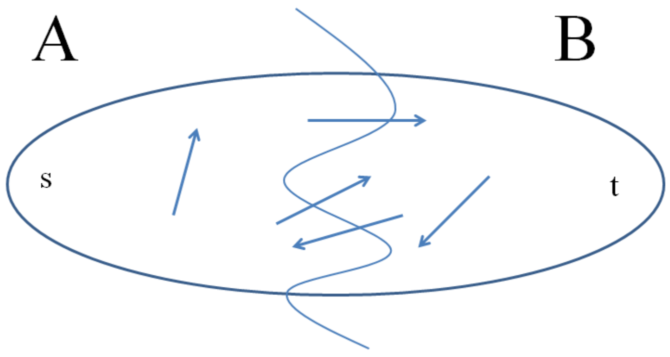


Figure 9: Visualization of a cut. Recall that we can partition edges into 4 categories:(i) edges with both endpoints in A; (ii) edges with both endpoints in B; (iii) edges sticking out of B; (iv) edges sticking into B.

Define $B = V - A$. Certainly s is in A, and hence not in B. (As we'll see, t isn't in B either.) We might have $B = \emptyset$ but this fine with us (we just want no vertices with excess in B).

The key trick is to consider the quantity

$$\sum_{v \in B} \underbrace{[\text{flow out of } v \text{ - flow in to } v]}_{\leq 0}. \tag{2}$$

[8]The argument bears some resemblance to the final step of the proof of the max-flow/min-cut theorem (Lecture #2) — the part where, given a residual network with no s-t path, we exhibited an s-t cut with value equal to that of the current flow.

Because f is a preflow (with flow in at least flow out except at s) and $s \notin B$, every term of (2) is non-positive. On the other hand, recall from Lecture #2 that we can write the sum in different way, focusing on edges rather than vertices. The partition of V into A and B buckets edges into four categories (Figure 9): (i) edges with both endpoints in A; (ii) edges with both endpoints in B; (iii) edges sticking out of B; (iv) edges sticking into B. Edges of type (i) are clearly irrelevant for (2) (the sum only concerns vertices of B). An edge $e = (v, w)$ of type (ii) contributes the value f_e once positively (as flow out of v) and once negatively (as flow into w), and these cancel out. By the same reasoning, edges of type (iii) and (iv) contribute once positively and once negatively, respectively. When the dust settles, we find that the quantity in (2) can also be written as

$$\sum_{e \in \delta^+(B)} \underbrace{f_e}_{\geq 0} - \sum_{e \in \delta^-(B)} \underbrace{f_e}_{=0}; \tag{3}$$

recall the notation $\delta^+(B)$ and $\delta^-(B)$ for the edges of G that stick out of and into B, respectively. Clearly each term is the first sum is nonnegative. Each term is the second sum must be zero: an edge $e \in \delta^-(B)$ sticks into A, so if $f_e > 0$ then the set A of vertices reachable by flow-carrying edges would not have gotten stuck as soon as it did.

The quantities (2) and (3) are equal, yet one is non-positive and the other non-negative. Thus, they must both be 0. Since every term in (2) is non-positive, every term is 0. This implies that conservation constraints (flow in = flow out) hold for all vertices of B. Thus all vertices with excess are in A. By the definition of A, there are paths of reverse edges in the residual network from these vertices to s, as desired. ■